RCU（Read Copy Update）

传统的锁（低效率令人难以接受）

- 读写都用一把锁
  改进：读写锁
- 读者并行，写不能，读写不兼容
  缺陷：由于缓存一致性，导致成本实际上是O(n^2)【N 个锁，平均N/2失败，每次N-1个CPU都要重置】获取锁的成本变得极其高【这个级别的成本可能会无效任何并行带来的收益】（即使是r_lock，他也带了一次写）

启发：要访问共享数据的时候，要尽可能的避免写操作。
接下来我们看一下存在数据写入者时的三种需要解决的场景（假设在修改链表）：

- 首先是数据的写入者只修改了链表元素的内容，将链表元素中的字符串改成了其他的字符串【RCU禁止，转化为下两个问题】
  - 修改为新建一个节点，然后原子替换（committing write）原来的，这就消除了一半新一半旧的可能。
  - 之前的节点会指向新节点，但是原来的节点不会修改他的next指针
  - 顺带一提，双向链表对于RCU极其不友好，单向的数据结构比较合适。
- 第二种场景是数据写入者插入了一个链表元素。
- 第三种场景是数据写入者删除了一个链表元素。【修改的时候也会有删除的操作】
  - 数据写入者到底要等待多久才能释放E2?

基本思想：让数据写入者变得慢一些。而且除了锁以外它还需要遵循一些额外的规则，从而让数据读取不需要锁，不需要写内存。

1. 数据读取者不允许在context switch时，持有一个被RCU保护的数据（也就是链表元素）的指针（不能返回指向原来元素的指针，只能返回原来数据的拷贝。）所以数据读取者不能在RCU critical 区域内出让CPU。
2. 数据写入者，它会在每一个CPU核都执行过至少一次context switch之后再释放链表元素。

RCU写入的基本过程

- 首先完成任何对于数据的修改
- 之后调用实现了上面第二条规则synchronize_rcu函数（要花费不少时间，可能要将近1个毫秒）
- 最后才是释放旧的链表元素
  对于RCU保护的数据来说，写操作相对来说较少，写操作多花费点时间对于整体性能来说不会太有影响。【RCU是针对多读少写的。】
  当然也存在另一种函数帮你提高写的速度，如果你真的很在意的话。（对于数据写入者不想等待的场景，可以调用另一个函数call_rcu，将你想释放的对象和一个执行释放的回调函数作为参数传入，RCU系统会将这两个参数存储到一个列表中，并立刻返回。之后在后台，RCU系统会检查每个CPU核的context switch计数，如果每个CPU核都发生过context switch，RCU系统会调用刚刚传入的回调函数，并将想要释放的对象作为参数传递给回调函数。这是一种避免等待的方法，因为call_rcu会立即返回。但是另一方面不推荐使用call_rcu，因为如果内核大量的调用call_rcu，那么保存call_rcu参数的列表就会很长，这意味着需要占用大量的内存，因为每个列表元素都包含了一个本该释放的指针。在一个极端情况下，如果你不够小心，大量的调用call_rcu可能会导致系统OOM，因为所有的内存都消耗在这里的列表上了。所以如果不是必须的话，人们一般不会使用call_rcu。）

- 由于现代硬件的执行顺序不确定性，所以RCU的指令应该自带BARRIER

一些启发

- 最有效的方法就是重新构造你的数据结构，让一些共享的变成不共享。
- 某些时候你又的确需要共享的数据，但是这些共享数据并没有必要被不同的CPU写入。（比如lab里面重构free list使得每个CPU核都有了一个专属的free list）将一个频繁写入的数据转换成了每个CPU核的半私有数据。这样大部分时候CPU核不会与其他CPU核的数据有冲突（还有个例子就是每个CPU的局部计数器）
  RCU能工作的核心思想是为资源释放（Garbage Collection）增加了grace period，在grace period中会确保所有的数据读取者都使用完了数据。所以尽管RCU是一种同步技术，也可以将其看做是一种特殊的GC技术。

```
// list reader:

  rcu_read_lock()
  e = head
  while(p){
    e = rcu_dereference(e)
    look at e->x ...
    e = e->next
  }
  rcu_read_unlock()

// replace the first list element:

  acquire(lock)
  old = head
  e = alloc()
  e->x = ...
  e->next = head->next
  rcu_assign_pointer(&head, e)
  release(lock)

  synchronize_rcu()
  free(old)
```

数据读取位于rcu_read_lock和rcu_read_unlock之间，这两个函数几乎不做任何事情。rcu_read_lock会设置一个标志位，表明如果发生了定时器中断，请不要执行context switch，因为接下来要进入RCU critical区域。所以rcu_read_lock会设置一个标志位来阻止定时器中断导致的context switch，中断或许还会发生，但是不会导致context switch。rcu_read_unlock会取消该标志位。所以这是一个集成在RCU critical区域的计数器。